SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE, VOL. 9, 103–124 (1997)

Practice

MSE: A Methodology for Software Evolution

VÁCLAV RAJLICH[1]
[1]*Department of Computer Science, Wayne State University, Detroit MI 48202, U.S.A.*

SUMMARY

Every program must continuously evolve, or it will become obsolete. This paper explores a methodology for software evolution within the setting of object-orientated programming. The methodology is based on the top–down propagation of change, and it is remotely related to stepwise refinement. To present the methodology, this paper uses one small example (Gregorian calendar) and one medium-sized example (calendar maintainer). This paper also explores an algorithm for scheduling object classes for update, and introduces a tool, 'Ripples', which helps programmers work with the process of software evolution. © 1997 by John Wiley & Sons, Ltd. J. Software Maintenance 9: 103–124, 1997.

(No. of Figures: 23. No. of Tables: 0. No. of Refs: 14.)

KEY WORDS: software evolution; object-orientated programming; software maintenance; program dependencies; ripple effects; software change

1. INTRODUCTION

It is a well-known fact that software has to evolve constantly or it will become obsolete. Reasons for that were reported by Lehman (1989) and by Parnas (1979), and can be summarized in the following way: software is not an isolated artefact, but operates within a wider context, being a part of a larger system. That system constantly changes. Whenever new software is introduced into a wider context, it changes that context and hence different software may be needed in the changed context. For example, suppose that a new software tool is successfully introduced into the work processes of a programming team. The programmers using that tool start doing things differently, and soon request a still different tool or modifications to current tools, or both. The introduction of the new software tool changed the context, and this altered context requires either modified tools or even completely different tools.

Another reason for software evolution is unanticipated uses of programs. For example, software implementations of spreadsheets were introduced to facilitate doing 'what if' types of calculations. Nowadays computerized spreadsheets are also widely used as form generators, and as interfaces to databases. Hence a new and completely unexpected set of requirements was placed on the software implementing the spreadsheets. Spreadsheets are examples of where requirements have changed substantially during the life of software.

CCC 1040–550X/97/020103–22$17.50

Received 30 July 1996
Revised 2 December 1996

If the usefulness or value of the software is to survive such changes, the software has to be modified. This paper terms as 'software evolution' the changes in software requirements that result in consequent changes of the software. In the past, the attention of software personnel has been concentrated on program development methodologies. These methodologies formalize implementation of new software. They provide a rigorous process for the requirements specifications and design. In that area, an extensive amount of work has been done, with a great number of development methodologies available to a programmer today—see for example, the overview paper of Monarchi and Puhr (1992).

This methodology situation contrasts with software evolution. Although evolving software is one of the most common software processes, most software personnel do it intuitively. The recognition of software evolution in the work of Jacobson (1993) and of Parnas (1979) deals mostly with evolutionary changes which can be anticipated during the design stage of development. However, many evolutionary changes cannot be or are not anticipated then. Lubars *et al*. (1992) studied how changes impact requirements specifications, but did not address the changes in the actual program software. This paper concerns itself with the impact of the changes on the software itself.

The purpose of this paper is to present a methodology for software evolution, abbreviated MSE, in the context of object orientation. For convenience in exposition, the paper uses examples for presenting MSE. A specialized version of MSE for an orthogonal architecture was presented by Rajlich and Silva (1996). This paper deals with more general software architectures and presents MSE within the general setting of object-orientated programming, using C++ as the example language.

Section 2 presents a simple example of MSE. Section 3 then summarizes the MSE process. The Appendix contains a medium-sized example of MSE involving a calendar maintainer. Section 4 offers a description of the tool 'Ripples' which supports MSE, including an algorithm for scheduling object classes for update. Section 5 presents the conclusions.

2. AN EXAMPLE: MILLENIUM UPDATE

2.1. Program for Julian calendar

As an example of evolution, consider a calendar program which operates on dates of this century (called Julian calendar). It will be evolved into a more general program which operates on the full range of the Gregorian calendar. Our starting program has two classes, 'day' and 'date'. Class 'day' of Figure 1 supports implementation of a day (Monday to Sunday). Its member functions determine the day based on the ordinal number of the day in the week, and output the result.

Class 'date' of Figure 2 represents any date of this century as a triple of two-digit integers separated by slashes, for example 02/16/96. Its member function *inputDate()* reads a date and checks its correctness. Its member function *findDistance()* provides the ordinal number of the day in the week, by calculating the number of days from the beginning of the century and returning this number modulo seven. The member function *length(int,int)* of Figure 3 returns the length of a month for a particular month and year. Figure 4 shows the function *main* of the program.


```
class day {
  public:
        void findDay(int);
        void printDay();
protected:
        char* DayName;
};
```

Figure 1. Class 'day'

```
class date {
public:
        int findDistance();
        void inputDate();
  protected:
        int Dd, Mm, Yy;
        int length(int, int);
};
```

Figure 2. Class 'date'

This program will become obsolete on 31 December 1999 (or before—depending on its use). In order to update it, let us evolve it into a program which calculates a day of any four-digit year within the Gregorian calendar—i.e., any date between the beginning of the Gregorian calendar and 31 December 9999. This update is here called a 'Millenium update', and it will give the program a sufficient new lease of life.

2.2. Changes in function *main*

The process of MSE evolution is a step-by-step top–down process, starting with the top objects. In this C++ example, the top object is just the function *main* and hence it is the first component of the program to be visited. The new function *main* is in Figure 5. The functions which need a change are renamed and highlighted in **boldface** for easy understanding.

Only two functions need a change. They both are members of the class 'date', which is the focus of the next step. The class 'day' does not need an update, because the week in the new program is still the same as in the old.


```
int date::length(int p_Mm, int p_Yy) {
        switch(p_Mm) {
                case 4: case 6: case 9: case 11:
                        return(30);
                case 2:
                        if ((p_Yy%4 == 0) && (p_Yy != 0))
                                return(29);   // February of a leap year
                        else
                                return(28);
                default:
                        return(31);
        }
}
```

Figure 3. Function date::length

```
main() {
        date Date;
        day Day;
        Date.inputDate();
        Day.findDay(Date.findDistance());
        Day.printDay();
}
```

Figure 4. Function main

2.3. Changes in class 'date'

The update of 'class date' starts with the creation of the 'class update graph' of Figure 6. It contains the relationships between the old and new members of the class. Whenever a new class member is derived from an old one by a change, there is a line between them. The symbol '*' means that the member was adopted from the old class into the new class without a change. The updated class 'date' is in Figure 7, with all updated class members highlighted in **boldface**.


```
main() {
        date Date;
        day Day;

        Date.inputNew();
        Day.findDay(Date.distanceNew());
        Day.printDay();
}
```

Figure 5. New function main

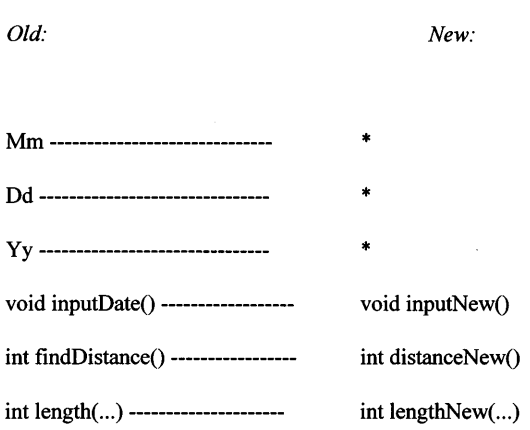

Figure 6. Class update graph of class 'date'

```
class date {
public:
        int distanceNew();
        void inputNew();
 protected:
        int Dd, Mm, Yy;
        int lengthNew(int p_Mm, int p_Yy);
};
```

Figure 7. Updated class 'date'

The new function *lengthNew()* of Figure 8 is the updated old function *length()*. It has to take into account all irregularities of the Gregorian calendar (Gregorian Calendar, 1983), namely:

- year divisible by four with a zero remainder is a leap year,
- year divisible by 100 with a zero remainder is an ordinary year,
- year divisible by 400 with a zero remainder is a leap year, and
- year divisible by 4 000 with a zero remainder is an ordinary year.

Similarly, other member functions of the class 'date' are updated. Please note that the process of update progresses in a top–down fashion, starting with the top function *main* and propagating through dependencies among the classes and class members (functions and data). In each instance, the new requirements are mapped onto the old class members, and then the appropriate code update is made. This process is a distant relative of stepwise refinement (Wirth, 1971; Rajlich, 1994) and it is described in the next section in more detail. The Appendix contains a larger example of the application of MSE.

```
int date::lengthNew(int p_Mm, int p_Yy) {
        switch(p_Mm) {
                case 4: case 6: case 9: case 11:
                        return(30);
                case 2:
                        if ((p_Yy%4 == 0) && (p_Yy%100 != 0) ||
                        (p_Yy%400 == 0) && (p_Yy%4000 != 0))
                                return(29); // February of a leap year
                        else
                                return(28);
                default:
                        return(31);
        }
}
```

Figure 8. Updated function date::lengthNew

3. STEPS OF MSE

3.1. Four steps

Programming-in-the-large is characterized by the granularities of entities (functions, class members, global variables, etc.), modules (classes, packages), files, and subsystems (interrelated classes, directories, etc.). The core constructs of programming-in-the-large in C language are functions and files. In C++, there are additional constructs like classes, class members, etc. Design methodologies focus mostly on programming-in-the-large, and so does MSE. Constructs of programming-in-the-large are interrelated by dependencies-in-the-large, and the change propagates from one construct into another through them.

During evolution by MSE, change always starts in the top objects of the dependency relationships among the classes—i.e., with the classes which do not support any other class. (In class dependency relationships, this paper terms a class as a 'support' class if it supplies data to or performs a function expected by some other class. This paper terms a class as a 'dependent' class if it receives data from some other class, or requires some other class to perform one or more functions for it.) This starting-from-the-top follows from the fact that evolution is a change in specifications, and that all specifications are tied together in the top objects, although parts may be delegated to supporting objects. The change propagates top–down to the functions and data of the supporting classes and their objects. In this sense, the MSE process resembles stepwise refinement applied to objects (Wirth, 1971; Rajlich, 1994). It differs from a 'change-and-fix' process where the change is made somewhere in the middle of the system, and the ripple effects of the change propagate both up and down through the system.

During the evolution process, the system has inconsistent dependencies: while the higher classes in the dependency relationships have been updated, some of the supporting classes have not been updated yet and hence embody the inconsistent dependencies. 'Scheduled classes' are the classes next in line for being changed to eliminate inconsistencies. They are members of the set of support classes which still have inconsistencies with one or more dependent classes. MSE proceeds in steps, each consisting of the use (often iterative) of the following phases:

- choosing a scheduled class,
- understanding the basis for the inconsistency,
- updating the scheduled class to eliminate the inconsistency, and
- verifying the accuracy of the updating.

3.2. Choosing a class

There may be several scheduled classes in the system, and one of them is chosen for the next update. Various criteria may be employed in this process. For example, they are risk control (choose the most risky class), or easy choice (choose the class whose modification will be easiest and will not likely propagate beyond the class), and so on (Rajlich, 1994). A special role among the criteria is played by the 'process cohesion criterion', which is the basis for tool 'Ripples' (see Section 4).

3.3. Understanding the basis for the inconsistency

There is an inconsistency between the chosen class and the classes dependent on it. The reason for the inconsistency is that the old specifications of the chosen class no longer fulfill the new requirements which the dependent classes place on it. This inconsistency must be understood, and a new set of specifications for the chosen class must be created. An example of a pair of old and new specifications (in plain English) are in Figures 15 and 16. Another pair are shown in Figures 20 and 21.

3.4. Updating the class

The understandings built in the prior phase serve as the basis for the phase of updating the scheduled class. The updating changes the class to conform to a new set of specifications. The change is done to each class member separately. There could be members which disappear in the modification, because they are no longer needed under the new requirements. There can be new members which have to be created from scratch, because some new requirements are so different from the old ones that the old members cannot be successfully modified to meet them. There are members which transfer to the new system without a change, representing the requirements shared by both the old and the new system. Finally there are members which need to be changed, because the requirements they represent changed slightly, and hence the members can be modified to satisfy the new requirements.

In the updating phase, a useful tool is the class update graph of Figure 9, which contains mappings between members of the old class and members of the new class. The members are usually functions. In the graph, the left column lists members of the old class, and the right column lists the members of the new class. The line between an old member and new member means that the new member is a modified old member. An example is *function f()* which was changed into function *fNew()* by small modifications. The modified function has a different name, to emphasize the change in functionality. Function *foobar()* is an example of a class member which disappeared in the change. Function *newFun()* has been created from scratch. Function *void foo()* was transferred from the old to the new program without a change, denoted by an asterisk. By such mapping, the programmer/analyst divides the change of class into smaller and more manageable parts. It is then much easier to change each class member individually in separation, following the requirements of change.

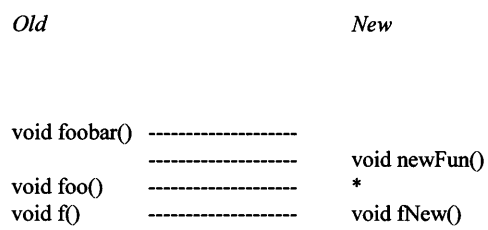

Figure 9. Generic form of class update graph

3.5. Verifying the accuracy

After the class update is complete, it can be verified and validated. Included in that process is the verification of any changes to the set of inconsistent dependencies and the set of scheduled classes. A system with inconsistent dependencies and scheduled classes can be verified through inspections, walkthroughs, or through testing with stubs for scheduled classes using standard stubbing techniques.

3.6. Context considerations

It should be mentioned that this paper deals with an ideal process. In practice, the programmer/analyst must look ahead—i.e., look beyond the immediate class—and as emphasized in the understanding phase, anticipate what ripple effects are going to be caused by the modification. The programmer/analyst has to avoid unmanageable ripple effects. Also, there may be a need for backtracks in the process, whenever an earlier modification proves to be an incorrect one. The programmer/analyst's work in doing MSE processes can be aided by tools, such as the tool 'Ripples' described in Section 4.

The Appendix contains an evolution of a program 'interactive library system' into 'interactive calendar maintainer for a medical clinic'. On first sight, it would seem that the two programs are so different that it does not make sense to evolve one into the other. However, if the evolution is done systematically, it is highly effective. There is enough commonality between the two programs that the code of the first one is changed into the second one with mostly small changes.

4. TOOL 'RIPPLES'

4.1. Role of the tool

The tool 'Ripples' described in this section supports MSE processes. It helps the programmer/analyst user to keep track of the changes in the software. The underlying algorithm of the tool schedules the classes for update, and is explained with the help of the following definitions.

4.2. Algorithm of 'Ripples'

Let C be a set of classes, and D a set of dependencies, such that $D \subseteq C \times C$. 'Top' is any class t such that for no $a \in C$, $\langle a,t \rangle \in D$. For simplicity, assume there is only one such class. Let I be a set of inconsistent dependencies, such that $I \subseteq D$. Then 'program' is a quadruple $P = (C,D,t,I)$. An evolution process S is a sequence $\langle S_1, \ldots, S_n \rangle$ such that $S_1 \cup \ldots \cup S_n = C$, where $S_1, \ldots, S_n$ are mutually disjoint sets, and $\langle a,b \rangle \in D$, $a \in S_i$, and $b \in S_j$ implies $i \leqslant j$. Then $S_1, \ldots, S_n$ are the steps of the process.

The previous definition is illustrated in the following lemma.

Lemma 1

All cycles in the dependency relation are always contained in one step.

If all cycles in the dependency relation are always contained in one step, then without a loss of generality they can be considered to be one single entity. In other words, MSE always updates all classes with a cyclical dependency in one step. In order to simplify the exposition of the algorithm below, assume that the graph of dependencies is a graph without cycles, and the nodes of the graph are either single classes or several classes in a cyclical relationship. For simplicity, the nodes of the dependency graph are called 'classes'.

Define the following sets for every $a \in C$:

$previous(a) = \{\langle b,a\rangle \mid \langle b,a\rangle \in D\}$
$following(a) = \{\langle a,b\rangle \mid \langle a,b\rangle \in D\}$
$previous^*(a)$ is the transitive closure of $previous(a)$
$scheduled(P) = \{a \in C \mid$ there exists $\langle c,a\rangle \in I\}$
$selected(P) = \{a \in scheduled(P) \mid$ for every $\langle b,a\rangle \in D$, $previous^*(b) \cap I = 0\}$

The set $selected(P)$ is a set of all classes which have an inconsistent dependency with at least one class depending on them, but there are no other inconsistent dependencies further up in the graph. The following lemma illustrates the situation.

Lemma 2

For dependency relation D without cycles, I is empty, if and only if $selected(P)$ is empty.

Let predicate $propagate(a)$ be true when class a is changed during the step of evolution to such a degree that it requires changes in supporting classes. Let it be false when class a is not changed or the change does not require change in the supporting classes.

The algorithm which schedules classes for update is in Figure 10 and it is in fact a variant of topological sort. The rule of class selection expressed in this algorithm is called 'process cohesion' (Rajlich, 1994). It makes sure that all new requirements of the class are known, before the class is updated. In this way, it avoids unnecessary repeated visits to the class.

Please note that MSE methodology is effective only when the graph of dependencies is either without cycles, or where the cycles are small. Only under these conditions, MSE divides the change propagation into many small steps and offers an effective process. Most systems are implemented in such a way that they satisfy this condition.

4.3. Implementation of 'Ripples'

The tool 'Ripples' is based on the scheduling algorithm described above (Vobilisetti, 1996). It guides the user in the evolution process by providing access to the selected classes. 'Ripples' presents the program dependencies graphically. After every step of evolution, it updates the graph of dependencies, the set of inconsistent dependencies, and the set of selected classes.


```
if (propagate(t))
        I = following(t);
else
        I = 0;
while (I != 0) {
        choose a ∈ selected(P);
        if (propagate(a))
                I = (I - previous(a)) ∪ following(a));
                        // propagates ripple effect
        else
                I = I - previous(a);
                        // stops ripple effect
}
```

Figure 10. Algorithm for scheduling classes for update

The input to 'Ripples' is a file containing a list of all source code files, both header files and definition files, and their paths. 'Ripples' parses these files, extracts the set of classes and their dependencies, and displays the graph of dependencies graphically. Classes are displayed as rectangles, dependencies as lines, and selected classes are represented as dashed rectangles. An example is shown in Figure 11 in the Appendix.

Upon choosing one of the selected classes, the user has a choice of updating it or leaving it unchanged. If the user chooses to leave it unchanged, the class will be marked as 'visited' and deleted from the set of selected classes. If the user chooses to update the class, an editor is invoked and the user can update the class. After the update, 'Ripples' parses the code again and displays the new graph of dependencies. The updated class is marked 'visited' and deleted from the set of selected classes. A new set of inconsistent dependencies and selected classes is then created based on the algorithm of Figure 10.

5. CONCLUSIONS

Software evolution, vertical reuse, and perfective maintenance are three variants of the same process. Understood in this context, software evolution contributes a substantial part of all software costs. According to Lienz, Swanson and Tompkins (1978) perfective maintenance is 60% of maintenance cost, and according to Carrol (1988) maintenance is 80% of the total cost of software. Hence, based on these combined data, approximately 50% of the software cost is in software evolution. The likely figure may be higher

because perfective maintenance is presumably more common today than it was in 1978, and evolution is also done during software development.

Software evolution can be done either intuitively or methodically. Therefore it is important to study the process of evolution, and develop methodologies and tools supporting it. The current 'change and fix' evolution prevails, partially because there has been very little research into evolution methodologies.

This paper presents a systematic approach, embodied in MSE methodology, for software evolution. The methodology is suitable for object-orientated software where the class dependency graph is an acyclic graph, or where the cycles in the dependencies are short. MSE can be summarized in the following way:

- Start the process with the top classes and their objects, propagating changes down through the dependency relations among classes—i.e., from the classes depending upon other classes to those dependent upon supporting classes. A class is scheduled for a change if it supports a class which has been already changed, and the existing dependency is inconsistent with the accomplished change.
- For each class which is to be changed, create a class update graph which maps the new requirements onto the old class members (usually functions). In the class update graph, four different situations arise:
    1. Some members disappear in the modification, because they are no longer needed under the new requirements.
    2. Some new members have to be created from scratch, because new requirements are so different from the old ones that the old members cannot be successfully modified to meet them.
    3. Some members transfer to the new system without a change, representing the requirements shared by both the old and the new system.
    4. Finally, some members need to be modified, because the requirements they represent have changed slightly, and hence the change can be satisfied by making modifications.
- Make the changes.
- After the class has been changed, update the set of scheduled classes.
- Verify that all changes and updates are made correctly.

This kind of methodology allows systems to be evolved over large domains. When evolution is practised intuitively, programmers usually resist evolving systems 'too far'. The effectiveness of MSE is substantially larger than the effectiveness of developing a new system from scratch, even in the situation where every class has to be changed. For example, Rajlich and Silva (1996) found that the effort required by evolution was approximately 40% of the estimated effort required to develop the system from scratch. In a case study reported by Vobilisetti (1996), an interactive library system was evolved into an interactive appointment system (see the Appendix to this paper). The errors in the new system can be mostly categorized as typing errors. There were no logical errors at the conceptual level. The use of MSE was instrumental in this result. The tool 'Ripples' supports scheduling classes for update, so that unnecessary backtracks are avoided.

A systematic way of handling software evolution opens a way to 'growing software'

as foreseen by Brooks (1987). In that case, there is no longer a distinction between prototyping and development, and systems are developed in a sequence of evolutionary changes.

APPENDIX

A.1. Introduction

This appendix contains an example of program evolution (Vobilisetti, 1996). The starting program implements an interactive library system, and the target program is to implement an interactive calendar maintainer for a medical clinic. The classes are annotated in the style of Rajlich, Doran and Gudla (1994). A more complete description of this case study can be found at: http://www.cs.wayne.edu/~vip/ProgramEvolution

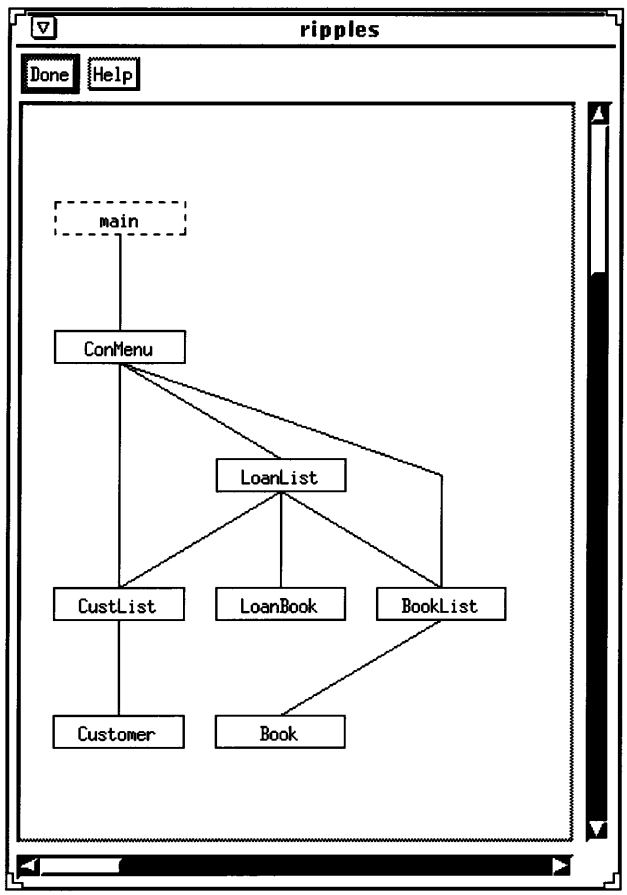

Figure 11. Screen of the tool 'Ripples'

A.2. Interactive library system

The original program is designed to maintain the status of books in a library system. It allows library customers to check out and return a book. It displays books checked out by a specific customer. It also displays all the books and the customers. The following are the requirements for the system:

1. To check out a book it needs the bar code number of the book and the library card number of the customer.
2. To return a book it needs the bar code number of the book.
3. To display the list of books checked out by a customer it needs the customer's library card number.
4. To display all the books/customers it needs no input.

A book can be checked out only by one customer at any time, and can be checked out by a different customer only after its return by the previous customer. A book can be returned only if the book is marked 'checked out' at the time of returning.

```
ConMenu {
        public :
                void DisplayMenu();
                int GetChoice();
                void ExecuteChoice();
        protected :
                int GetCustCardNum();
                int GetBookBarCode();
                int IsInteger(char *);
                int choice;
                CustList custlist;
                BookList booklist;
                LoanList loanlist;
        };
```

Figure 12. Old class definition for class 'ConMenu'


```
class conMenu {
        public :
                void displayMenu();
                int getChoice();
                void executeChoice();
        protected :
                char* getDocName();
                int getPatSSN();
                int isInteger(char *);
                int choice;
                docList DocList;
                patList PatList;
                appmnt Appmnt;
};
```

Figure 13. New class definition for class 'ConMenu'

A.3. Interactive calendar maintainer for a medical clinic

The library system is evolved using MSE into a system to maintain the appointment schedules for doctors in a medical clinic. The two application domains are totally different, but there are sufficient similarities to make the evolution feasible. All the classes except function *main* need to be updated. The evolution process takes seven steps, one for each class.

This system allows the appointments nurse to make or cancel appointments for any doctor at a specified time and day. It displays all appointments in the current week for a specific doctor. It also displays all available doctors and the patients. A doctor can have a maximum of three appointments at 1:00 pm, 2:00 pm and 3:00 pm on the five regular working days (Monday to Friday) only. The following are the requirements for the system:

1. To make an appointment it needs the doctor's name, the patient's social security number, the time and the day of appointment.
2. To cancel an appointment it needs the doctor's name, the time and the day of the appointment. It requests the system user to inform the patient of the cancellation of the appointment.
3. To display appointments for a particular doctor it needs the doctor's name.
4. To display all the available doctors and patients it needs no input.

A.4. Function *main* and class 'ConMenu'

Function *main* is just a driver for the class 'ConMenu'. It is the first entity of the program to be visited, and it is left unchanged. Figure 11 shows the screen of the tool 'Ripples' in this situation.

Class 'ConMenu' is the second program entity to be visited. It implements the interaction with the user. The changes are concentrated in the input/output statements and the names for the function calls. The change is categorized under slight modification. Figures 12 and 13 contain the old and new interfaces of the class, respectively. The class update graph is in Figure 14. Figures 15 and 16 contain old and new class annotations, respectively.

A.5. Class 'LoanList'

In the old application, class 'LoanList' implements the loan relation between the customer and the book. The class maintains all the books checked out by customers in a

New members		*Old members*
displayMenu()	----------0----------	DisplayMenu()
getChoice()	----------	GetChoice()
executeChoice()	----------0----------	ExecuteChoice()
isInteger()	----------	IsInteger()
getDocName()	----------0----------	GetCustCardNum()
getPatSSN()	----------0----------	GetBookBarCode()
choice	----------	choice
DocList	----------0----------	custlist
PatList	---------o0o----------	booklist
Appmnt	---------o0o----------	loanlist

Explanations:

---0---	stands for slight modification
--o0o--	stands for heavy modification
-------	stands for no modification

Figure 14. Class update graph of class 'ConMenu'

This class provides the user interface for the whole application. It provides functions which display the menu options (check out book, return book, display list of books taken out by a particular customer, display all the customers, display all the books), reads the user response, executes the corresponding option.

Figure 15. Old domain annotation of class 'ConMenu'

This class provides the user interface for the whole application. It provides functions which display the menu options (make appointment, cancel appointment, display appointments for a particular doctor, display all doctors, display all patients), reads the user response, executes the corresponding option.

Figure 16. New domain annotation of class 'ConMenu'

linked list. The equivalent concept, in the new application, is to maintain all the appointments for the patients with the doctors along with time and day of the week. So, the class is selected for the visit and the class name was changed to 'appmnt' (appointment) to reflect the new domain. The data structure of the class 'LoanList' is changed from a linked list to a three-dimensional array wherein the first dimension represents the doctor, second dimension represents the day and the third dimension represents the time. Owing to the change in data structure of the class, the implementation of all the member functions are changed. Hence, the change is categorized under heavy modification.

Figures 17 and 18 contain the old and new interface of the class, respectively. The graph of class update is in Figure 19. Figures 20 and 21 contain old and new class

```
class LoanList {
    public :
        LoanList();
        ~LoanList();
        void CheckOutBook(int, int);
        void ReturnBook(int);
        void DisplayBooks(const CustList &, const BookList &, int);
    protected :
        LoanBook *head;
};
```

Figure 17. Old class definition of class 'LoanList'


```
class appmnt {
        public :
                appmnt();
                ~appmnt();
                void makeAppointment(char *, int);
                void cancelAppointment(int);
                void displayAppoint(const docList &, const patList &, int);
        protected :
                void getTimeAndDay(int &time, int &day);
                int app[NO_DOCTORS][NO_SCHEDULES][NO_WORKINGDAYS];
};
```

Figure 18. New class definition as 'appmt' for the old class 'LoanList'

New members		*Old members*
Appointment()	--------o0o-------	CheckOutBook()
cancelAppointment()	--------o0o-------	ReturnBook()
displayappointment()	--------o0o-------	DisplayBooks()
getTimeAndDay()	----------	
app	--------oOo-------	head

Figure 19. Class update graph of class 'LoanList'

This class maintains a list of pairs (bar_code, card_num) where bar_code is the bar code of the book checked out by the customer whose card number is card_num. It provides functions to check out and return books. It also provides function to display the list of books checked out by a particular customer.

Figure 20. Old domain annotation of class 'LoanList'

This class maintains the appointments of all doctors for the coming week (five working days). It provides functions to make and cancel appointments for a doctor. It also provides function to display appointments for a particular doctor for the coming week.

Figure 21. New domain annotation of class 'appmnt'

annotations, respectively. Figures 22 and 23 contain the old and new graphs of dependencies, respectively. Note at this point, the graph of dependencies consists partially of the old and partially of the new classes.

The class 'LoanBook' is dropped from the new application due to the change in data structure.

A.6. Class 'CustList'

There are two scheduled classes 'CustList' and 'BookList'. Class 'CustList' is chosen because it requires minor changes to meet the new requirements. In the old application, class 'CustList' maintains a list of customers in a sequential list (array). The equivalent concept in the new application, is to maintain a list of doctors. This class is given a new name 'docList' (list of doctors) to reflect the new domain. As all the changes are in the names (of type, variable, function, etc.) the change is categorized under slight modification.

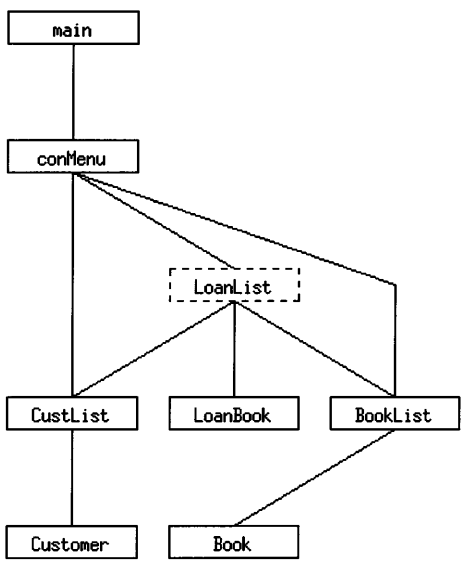

Figure 22. Graph of dependencies before update of class 'LoanList'

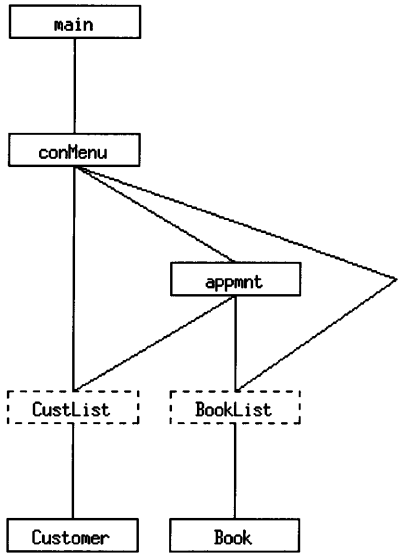

Figure 23. Graph of dependencies after update of class 'appmnt'

A.7. Class 'Customer'

There are two scheduled classes 'BookList' and 'Customer'. Class 'Customer' is selected in order to complete one of the subsystems in the architecture. In the old application, class 'Customer' represents the entity customer. As the class 'CustList' (list of customers) is changed to the class 'docList' (list of doctors), the concept corresponding to entity Customer in the new application will be the entity 'Doctor'. Therefore, the class is given the new name 'Doctor'. In the update process, a new data member is added and the names of some data and function members are changed to reflect the new domain. The implementations of the member functions are also changed. Owing to their extensiveness, the changes to 'Customer' are categorized as heavy modification.

A.8. Class 'BookList'

Class 'BookList' is the only scheduled class in the system. In the old application, the class 'BookList' maintains a list of books. The equivalent concept in the new application, is to maintain a list of patients. This class is given a new name 'patList' (list of patients) to reflect the new domain. Here, the changes are in the member names to represent the new domain and corresponding changes in the class function member definitions. Also, two new member functions are added and one old function member is dropped. The class 'Book' is moved to the list of scheduled classes. As there are major modifications made to produce the replacement for 'BookList', the change is categorized under heavy modification.

A.9. Class 'Book'

At this point, the class 'Book' is the only scheduled class left for the visit. In the old application class 'Book' represents the entity book. Because the class 'BookList' (list of books) was changed to the class 'patList' (list of patients), the concept corresponding to the entity 'book' in the new application will be the entity 'patient'. In the update process, the names of some data and function members are changed to reflect the new domain. Also one new member function is added. Owing to major changes, the change is categorized under heavy modification.

References

Brooks, F. P. (1987) 'No silver bullet', *Computer*, **20**(4), 10–19.

Carroll, P. B. (1988) 'Computer glitch: patching up software occupies programmers and disables systems', *Wall Street Journal*, **118**(14), A1.

Gregorian Calendar (1983), in *The New Encyclopedia Britannica*, Micropedia Vol IV, Encyclopedia Britannica, Inc., Chicago, IL, p. 725.

Jacobson, I. (1993) *Object-oriented Software Engineering*, Addison-Wesley, Reading, MA, 528pp.

Lehman, M. M. (1989) 'Uncertainty in computer application and its control through the engineering of software', *Journal of Software Maintenance*, **1**(1), 3–27.

Lienz, B. P., Swanson, E. B. and Tompkins, G. E. (1978) 'Characteristics of application software maintenance', *Communications of the ACM*, **21**(6), 466–471.

Lubars, M., Meridith, G., Potts, C. and Richter, C. (1992) 'Object-oriented analysis for evolving systems', in *Proceedings of the 14th International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos, CA, pp. 173–185.

Monarchi, D. and Puhr, G. (1992) 'A research typology for object-oriented analysis and design', *Communications of the ACM*, **35**(9), 35–47.

Parnas, D. L. (1979) 'Designing software for ease of extension and contraction', *Transactions on Software Engineering*, **5**(3), 128–138.

Rajlich, V. (1994) 'Decomposition/generalization methodology for object-oriented programming', *Journal of Systems and Software*, **24**(1), 181–186.

Rajlich, V., Doran, J. and Gudla, R. T. S. (1994) 'Layered explanations of software: a methodology for program comprehension', in *Proceedings of the 3rd IEEE Workshop on Program Comprehension*, IEEE Computer Society Press, Los Alamitos, CA, pp. 142–145.

Rajlich, V. and Silva, J. (1996) 'Evolution and reuse of orthogonal architectures', *Transactions on Software Engineering*, **22**(2), 153–157.

Vobilisetti, S. (1996) 'A tool for software evolution', MS Thesis, Department of Computer Science, Wayne State University, Detroit, MI, 81pp.

Wirth, N. (1971) 'Program development by stepwise refinement', *Communications of the ACM*, **14**(4), 221–227.

Author's biography:

Václav Rajlich is a professor of Computer Science at Wayne State University, Detroit, Michigan, U.S.A. Before that he was an associate professor at the University of Michigan. His current research interests are legacy software comprehension and evolution. He is a founder and former general chair of the IEEE International Workshop on Program Comprehension in 1992, 1993, 1994, 1996, and general chair of the International Conference on Software Maintenance in 1992 and 1997.